

A JAVA API FOR LOW-LEVEL SOCKET NETWORK ACCESS**Final Report****Period: December 15, 2004–June 13, 2005****Contract No. W15P7T-05-C-S202****Create Project 7406**

Richard W. Kaszeta
Principal Investigator

Eric M. Friets
Project Engineer

Distribution A: Approved for public release; distribution is unlimited.

SBIR Data Rights

Contract No.: W15P7T-05-C-S202
Contractor Name: Create Incorporated
Contractor Address: P.O. Box 71, Hanover, NH 03755
Expiration of SBIR Data Rights Period: 13 June 2010

The Government's rights to use, modify, reproduce, release, perform, display, or disclose technical data or computer software marked with this legend are restricted during the period shown as provided in paragraph (b) (4) of the Rights in Noncommercial Technical Data and Computer Software—Small Business Innovative Research (SBIR) Program clause contained in the above identified contract. No restrictions apply after the expiration date shown above. Any reproduction of technical data, computer software, or portions thereof marked with this legend must also reproduce the markings.

20050620 127

**Declaration of Technical Data Conformity**

The Contractor, Creare Incorporated, hereby declares that, to the best of its knowledge and belief, the technical data delivered herewith under Contract No. W15P7T-05-C-S202 is complete, accurate, and complies with all requirements of the contract.

Date 10 June 2008

Name and Title of Authorized Official  BUSINESS ADMIN MGR

DISCLAIMER

This report was prepared by Creare Inc. for the U.S. Army. Neither Creare, nor any person acting on its behalf, makes any warranty or representation, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of the information, apparatus, method or process disclosed in this report. Nor is any representation made that the use of the information, apparatus, method or process disclosed in this report may not infringe privately-owned rights.

Creare assumes no liability with respect to the use of, or for damages resulting from the use of, any information, apparatus, method or process disclosed in this report.

TABLE OF CONTENTS

1	INTRODUCTION.....	1
2	PROJECT SUMMARY	1
3	PHASE I OBJECTIVES AND ACCOMPLISHMENTS.....	1
3.1	TASK 1. RESEARCH IMPLEMENTATION OF RAW SOCKET API	1
3.1.1	Design Goals.....	2
3.1.2	Functional Layout.....	3
3.1.3	Security Considerations.....	4
3.1.4	JavaSock API Architecture	4
3.1.5	Result.....	4
3.2	TASK 2. DESIGN A PROTOTYPE API	4
3.3	TASK 3. REFERENCE IMPLEMENTATION	5
3.4	TASK 4. MANAGE AND REPORT	6
4	THE JAVASOCK LIBRARY AND API.....	7
4.1	THE JAVASOCK JAVA API.....	7
4.2	THE JAVASOCK NATIVE INTERFACE.....	9
4.3	THE JAVASOCK KERNEL SERVICE	9
4.4	JAVASOCK SECURITY.....	10
4.5	EXAMPLE APPLICATION.....	10
5	FUTURE PLANS	11

LIST OF FIGURES

Figure 1.	JavaSock Architecture, Showing Divisions Between Java and Native Code, as Well as Divisions Between User and Kernel Code.....	3
Figure 2.	A Sample Swing Application Using the JavaSock API.....	6

LIST OF TABLES

Table 1.	JavaSock Java API Abstract and Implemented Classes.....	8
Table 2.	The org.javasock Package Hierarchy.....	8

1 INTRODUCTION

This is the final report for this project that is being performed by Create Inc. for the U.S. Army. It covers the time period of 15 December 2004 to 13 June 2005. The overall goal of this project is to develop a Java API which uses a native code library allowing Java programs to manipulate raw sockets through a consistent and secure interface. This API allows Java programs to directly implement low-level networking sockets, such as packet filtering, packet sniffing and reading of low-level IP or transport layer data.

2 PROJECT SUMMARY

Modern military computer applications are increasingly using the Java computer language. While Java provides a versatile cross-platform computing environment with strong security and ease-of-use (java.net.Socket), this support is limited to traditional TCP/IP socket-style communications, and lacks the ability for the programmer to directly access lower levels of the network protocol stack, such as raw sockets. Create has developed a Java API which uses a native code library allowing Java programs to manipulate raw sockets through a consistent and secure interface. This API allows Java programs to directly implement low-level networking sockets, such as packet filtering, packet sniffing and reading of low-level IP or transport layer data. In Phase I, we researched implementing raw sockets in Java, reported on the feasibility and implications of such an implementation, and created a prototype API and an application that demonstrates a subset of this prototype API.

Below is a description of the work performed on Phase I during the reporting period.

3 PHASE I OBJECTIVES AND ACCOMPLISHMENTS

Project Schedule									
Tasks	Month								
	1	2	3	4	5	6	7	8	9
1. Research Implementation of Raw Socket API									
2. Design a Prototype API									
3. Reference Implementation									
4. Manage and Report									
Kickoff Meeting	♦								
Progress Reports		♦	♦	♦	♦		♦	♦	
Final Results Meeting						♦			
Final Report						♦			♦
5. Option Phase									

3.1 TASK 1. RESEARCH IMPLEMENTATION OF RAW SOCKET API

The objective of this task was to define and document the required features and functionality of the desired API.

Under this task, Dr. Richard Kaszeta and Mr. William Finger traveled to Fort Monmouth, NJ for a kickoff meeting with the technical monitor and staff. At this meeting, the proposed raw socket API (JavaSock) was presented and reviewed. Based upon discussions at the kickoff meeting, a target computing platform was chosen (Windows XP), and the desired functionality of the API and demonstration application was specified (a Java-based Ethernet and IP packet sniffer).

Based upon a review of existing software packages for providing low-level network socket access in Java applications, we decided to design our own Java library and low-level native interface. During the first three months of the project, we developed a plan for developing our new low-level socket API (JavaSock) by determining the design goals of the API, the functional layout and structure of the proposed API, the modular architecture of the API, and the security implications of our proposed API. The result of this research was issued as Creare Technical Memorandum TM-2440. However, the key conclusions from that report are included below.

3.1.1 Design Goals

The JavaSock API was developed with a number of important goals.

- **General Purpose:** JavaSock library needs to provide general purpose APIs that allow low-level protocol access at the DataLink, Network, and Transport levels of the OSI 7-layer Networking Reference Model. The library has been developed so that the API should be applicable for a variety of networking programming tasks, such as packet sniffing, TCP SYN scans, and network monitoring.
- **Security:** The JavaSock library is designed to provide access to low-level network communications, which is a sensitive operation. Raw network access, such as monitoring all the traffic on a network segment, provides a risk to the integrity and security of both the network (since many of the network security and data integrity mechanisms can be bypassed) and the computer (since low-level access to the network interface requires some level of administration privilege). The JavaSock library is designed to minimize these security risks, and provide access only when a user has appropriate administrative privileges.
- **Platform Independence:** The Java language provides an architecture-neutral programming environment, and therefore, the JavaSock library should be designed in a manner that allows it to be easily ported to other platforms.
- **Efficiency:** Due to the possibility of large amounts of network traffic being present when low-level networking access is desired, the JavaSock library is designed to operate efficiently, with minimal processing overhead.
- **Simplicity:** The JavaSock library is designed to be simple and easy to program for basic network operations, and avoid the need for extensive learning of a complex API or grammar.

3.1.2 Functional Layout

The JavaSock Raw socket API is structured into three components (see Figure 1, below): a Java language library, a native interface, and a kernel service module. The Javsock library (javasock.jar) includes a Java-class hierarchy following the OSI layered network approach: separate abstract Java classes are provided for access at the Transport (TCP), Network (IP), and DataLink (Ethernet) levels. Each class includes extensions which provide handlers for the appropriate network protocols for that layer (TCP and UDP handlers at the Transport level, IP/ICMP/ARP handlers at the Network layer, etc.) which attach to a lower layer network handler. At the bottom layer, a native Win32 DLL (javasock.dll) provides a native interface to the network interface, minimizing the amount of code that is platform-specific, as well as minimizing the interface between the native code and the Java classes. The network interface itself is provided as a minimal NT kernel service module.

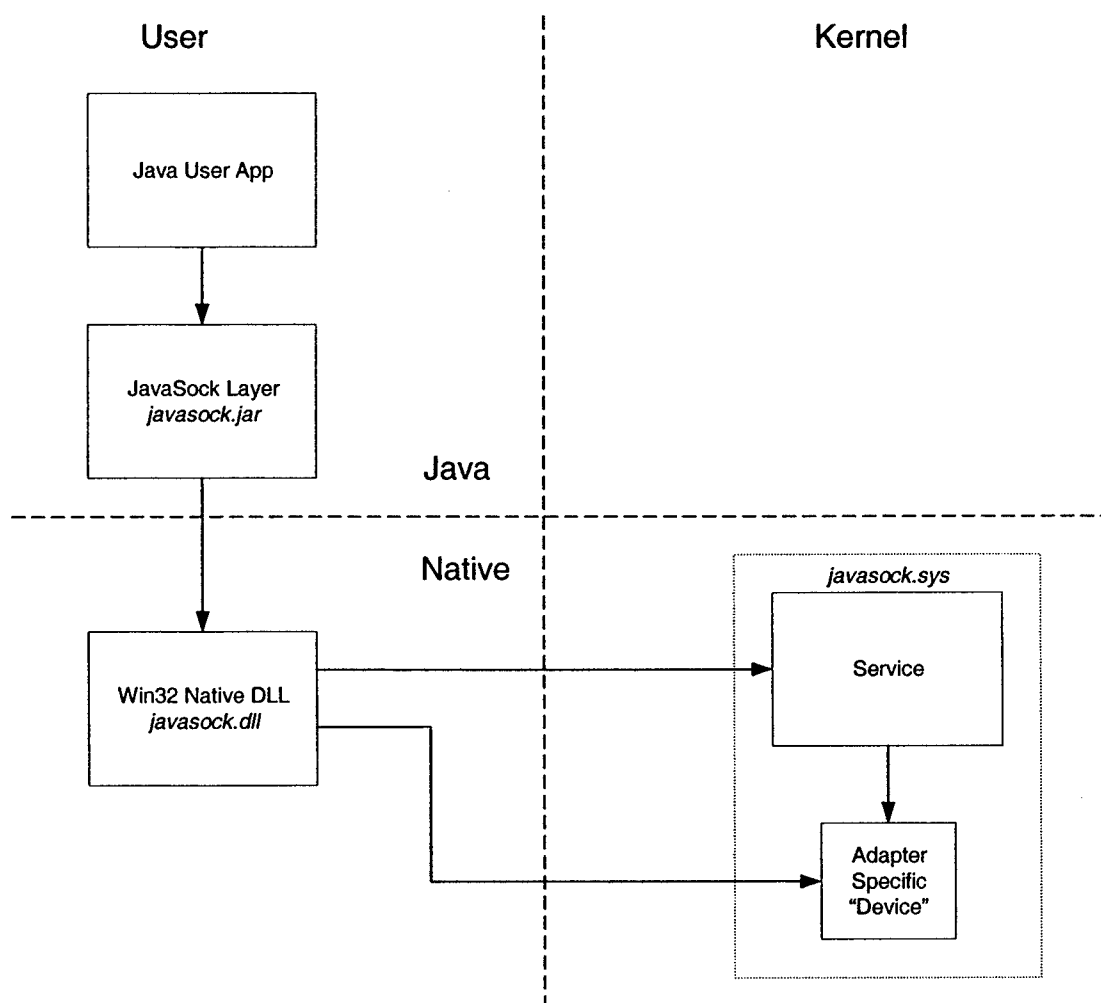


Figure 1. JavaSock Architecture, Showing Divisions Between Java and Native Code, as Well as Divisions Between User and Kernel Code.

3.1.3 Security Considerations

The security architecture of the JavaSock library makes use of the fine-grained security architecture available in Windows. The tightest level of security will involve the kernel level service, which intercepts incoming packets. During installation, which will require Administrator privileges on the machine, the service will be configured such that it can only be started by a user with Administrator privileges. This is performed using the Windows API function *SetServiceObjectSecurity*.

Subsequent connections to the service must originate from the same user level process that started it. This helps guarantee that an orphaned service is not used by a third party for malicious purposes. Since Administrator privileges were required to start the service, the same process should still have them while connecting. A process may make several connections to the service to open different adapters for configuration and capture.

Once data capture is complete, the user level process should stop the service. If this is not performed, the service will disable itself after an appropriate interval.

3.1.4 JavaSock API Architecture

The structure of the Java API provides powerful network monitoring capabilities from a simple model. The API handles the vast majority of low-level details, freeing the application writer to concentrate on their analysis. There are two groups of classes in the API, one that serves as handlers for layers in the OSI model, and one that serves as data objects for layers in the OSI model. The API defines an abstract class for each layer, and for its associated data object. Classes are implemented for common devices and data objects at each layer. The use of abstract classes forces a coherent structure on applications, while allowing ready extension to particular devices. It also hides all the native code that interfaces to machine specific devices.

A complete review of the JavaSock API architecture is presented in Section 4 below.

3.1.5 Result

The result of this task was a complete Design Report, issued in mid March 2005. With the issuing of this report, Task 1 was successfully completed.

3.2 TASK 2. DESIGN A PROTOTYPE API

The objective of this task was to design a prototype Raw Socket API, Java classes, and the native language libraries. As mentioned in Task 1, the JavaSock API consists of three modules: a Java class structure, a native interface, and a kernel service module. During the project, prototype versions of the Java class library (javasock.jar), native interface (javasock.dll), and the kernel service module (javasock.sys) were developed, the interfaces between the modules designed and implemented, and the resulting software tested for functionality. The software development approach used was incremental bottom-up development. The first component implemented was the NT kernel service module (javasock.sys), implemented initially as an Administrator-only secure, basic packet capturing service, followed by an extended version including secure authentication methods and kernel-level packet filtering.

Once the NT kernel service was written and tested, the Win32 native DLL (javasock.dll), which allows the NT kernel service module's services to be accessed through Java's Java Native Interface (JNI) mechanism, was written and tested, followed by the implementation and testing of the JavaSock Raw Socket library itself (javasock.jar).

The result of this task was a complete Java low-level socket library, consisting of three functional, tested modules. These modules serve as the foundation for the demonstration application developed in Task 3.

3.3 TASK 3. REFERENCE IMPLEMENTATION

The objective of this task was to develop a demonstration application that demonstrates some of the additional networking support of the API developed in Task 2, using a subset of the developed API.

In Task 1, it was determined that the demonstration application would be a Java-based packet sniffer, capable of demonstrating the basic operation of the JavaSock API, including basic capturing and logging of packet data at the Ethernet Frame, IP Packet, and TCP streams.

Since the JavaSock API is standard Java, it can easily be integrated with other Java toolkits to create functional raw-socket applications. The JavaSock demonstration application (screenshot shown in Figure 2) is a functional GUI packet sniffer, implemented using the prototype JavaSock libraries and the standard Java Swing graphics libraries.

Upon starting the JavaSock Demo application, it starts the kernel service and obtains a list of the available adapters. These adapters are shown in the panel on the upper left. After specifying the optional filters, IP and/or UDP, packets are captured by pressing the Start Capture button. Packets are displayed with their time of capture in the table at the bottom of the display. After capture is stopped, the packet log may be saved to a file for further analysis. This simple example serves to demonstrate the ease with which the network may be monitored and incoming packets filtered to focus on network traffic of interest.

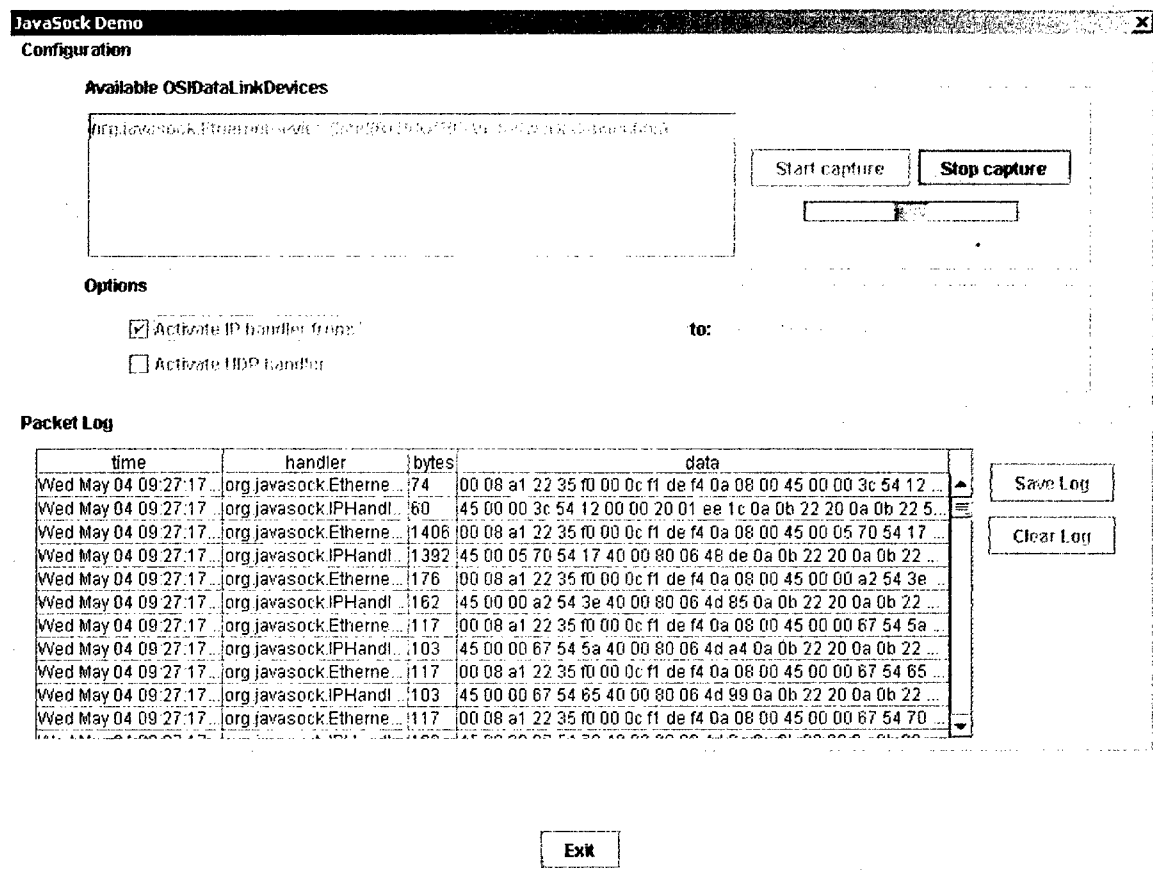


Figure 2. A Sample Swing Application Using the JavaSock API

The result of this task is the completion of a functional Java application that serves two purposes:

- Demonstrating the utility and functionality of the complete JavaSock API, and providing sample source code for using the API.
- Identifying possible technical issues, performance issues, and security implications of using the JavaSock API in an actual application.

The demonstration application, the JavaSock API, and support documentation and source code will be delivered at the completion of the contract, completing this task.

3.4 TASK 4. MANAGE AND REPORT

Under this task, we planned, monitored, and reported the technical progress and expenses of the project. Principal activities in this task include the formulation and upkeep of the project plan, allocation and management of technical staff, and preparation of monthly progress reports.

During the reporting period, five progress reports and a final report (this document) were prepared and submitted. Additionally, the Principal Investigator, Project Engineer, and Chief

Software Architect traveled to Ft. Monmouth, NJ to present the conclusions of the Phase I effort, as well as demonstrate the JavaSock API and the demonstration application.

With the issuing of the Final Report, this task will be completed.

4 THE JAVASOCK LIBRARY AND API

The JavaSock API is structured into three components (see Figure 1, above): A Java language library, a native interface, and a kernel service module. The JavaSock library (javasock.jar) includes a Java-class hierarchy following the OSI layered network approach: separate abstract Java classes are provided for access at the Transport (TCP), Network (IP), and Interface (Ethernet) levels. Each class includes extensions which provide handlers for the appropriate network protocols for that layer (TCP and UDP handlers at the Transport level, IP handlers at the Network layer) which attach to a lower layer network handler. At the bottom layer, a native Win32 DLL (javasock.dll) provides a native interface to the network interface, minimizing the amount of code that is platform-specific, as well as minimizing the interface between the native code and the Java classes. The network interface itself is provided as a minimal NT kernel service module.

Each component of the JavaSock API will be discussed below.

4.1 THE JAVASOCK JAVA API

The use of the JavaSock API in a program requires three distinct steps. First, a hierarchy of objects is constructed, which is a reflection of the desired OSI layers required by the application. Next, any required packet filters are added to the appropriate layers. Finally, callbacks must be established to handle matching packets.

The classes of the JavaSock API follow the layers in the OSI Network Model, and instantiated classes are associated with a corresponding instantiated class one layer lower in the model. This structure creates a simple model for both filter definitions and packet callbacks. Filtering is done by the kernel system component, for efficiency. No filtering is required, though, so every packet on the network can be grabbed for analysis.

The structure of the Java API provides powerful network monitoring capabilities from a simple model. The API handles the vast majority of low level details, freeing the application writer to concentrate on their analysis. There are two groups of classes in the API: one that serves as handlers for layers in the OSI model, and one that serves as data objects for layers in the OSI model. The API defines an abstract class for each layer, and for its associated data object. Classes are implemented for common devices and data objects at each layer. The use of abstract classes forces a coherent structure on applications, while allowing ready extension to particular devices. It also hides all the native code that interfaces to machine specific devices.

The relevant layers in the OSI model are DataLink, Network, and Transport. The DataLink layer controls the physical link of communication, and has a data unit of a frame. The Network layer has a data unit of a packet, and provides end-to-end communications. The Transport layer establishes connections and transfers data, using packets. Table 1 lists the

abstract and implemented classes, extensions, and filters, while Table 2 shows the Java hierarchy for the JavaSock API (implemented as the Java Package org.javasock). Additional extensions can be readily added for other devices, handlers, and data objects.

Table 1. JavaSock Java API Abstract and Implemented Classes				
OSI Layer Abstract Class	Implemented Extension(s)	OSI Layer Abstract Data Object	Implemented Extension(s)	Implemented Filter(s)
OSIDataLinkDevice		OSIDataLinkFrame		
	EthernetDevice		EthernetFrame	EtherTypeFilter
OSINetworkHandler		OSINetworkPacket		
	IPHandler		IPPacket	IPFilter
OSITransportHandler		OSITransportPacket		
	TCPHandler		TCPPacket	TCPFilter
	UDPHandler		UDPPacket	UDPFilter

Table 2. The org.javasock Package Hierarchy	
<p>CLASS HIERARCHY:</p> <p>java.lang.Object</p> <ul style="list-style-type: none"> o org.javasock.Filter <ul style="list-style-type: none"> o org.javasock.EtherTypeFilter o org.javasock.IPFilter <ul style="list-style-type: none"> o org.javasock.IPFromToFilter o org.javasock.MatchFilter o org.javasock.NullFilter o org.javasock.TCPFilter o org.javasock.UDPFilter o org.javasock.PacketHandler <ul style="list-style-type: none"> o org.javasock.OSIDataLinkDevice <ul style="list-style-type: none"> o org.javasock.EthernetDevice o org.javasock.OSINetworkHandler <ul style="list-style-type: none"> o org.javasock.IPHandler o org.javasock.OSITransportHandler <ul style="list-style-type: none"> o org.javasock.TCPHandler o org.javasock.UDPHandler o org.javasock.PacketStatistics <p>INTERFACE HIERARCHY :</p> <ul style="list-style-type: none"> o java.util.EventListener o org.javasock.PacketListener 	

There are no constructors for devices at the DataLink layer. A static method of OSIDataLinkDevice returns an array of devices that are running and accessible to the operating system. Constructors for the handlers require a reference to a handler or device to which it will be attached. Thus, the devices and handlers are structured in a tree that corresponds to the OSI levels and to the way data flows up and down the levels.

The tree of instantiated classes operates in two modes. In configuration mode, branches may be added to the tree, and filters may be set up. In operational mode, the structure and

filtering are fixed, and data from the network that matches the filters are passed up the tree for logging and analysis.

Filtering is set up at each branch of the tree. For example, to monitor traffic between two hosts, a filter would be set in the IPHandler class. Each branch also sets up a generic filter, which checks for the appropriate type of frame or packet. In the case of IPHandler, only frames containing IP packets as payload would be passed to it.

To begin filtering and capturing packets, the OSIDataLinkDevice startCapture method is called. The tree is traversed, and the filters accumulated. The filters are converted to a simple and secure binary format, described below, that the kernel service can efficiently evaluate. The developer need know nothing about the details of this binary format, they simply specify the filters.

When the kernel service identifies a frame that matches the filters, it passes the frame to the native DLL, which in turn issues a callback to the Java API. The callback includes which handlers the filtering matched, so the filtering does not have to be repeated. Each node in the tree issues callbacks as appropriate. These callbacks are issued by the abstract class, so the developer need only implement the callback method, in which they perform whatever analysis or logging is appropriate to the task at hand. If the kernel service is unable to sustain the throughput, packets are dropped.

4.2 THE JAVASOCK NATIVE INTERFACE

Directly below the Java layer is the native shared library. This module serves as the interface logic between Java and the kernel level code. This code will be platform dependent, but the native interface presented to the Java layer will be identical for all platforms. The Windows implementation, *javasock.dll*, will start the JavaSock service, forward the list of adapters to the Java layer, and make connections to the service to support packet interception for one or more adapters. It will also handle issues which arise in managing memory between the service and Java.

4.3 THE JAVASOCK KERNEL SERVICE

The Java Sock Kernel Service performs packet interception and filtering as requested by the user. The Windows implementation will reside in *javasock.sys*. The code in this module determines the list of network adapters available on the machine, and provides it to the user level module. It provides a means for the user level code to open and request packets, with optional filtering, from one or more adapters. It will filter out packets which do not meet the inclusion criteria specified by the user application, in as rapid a means as possible. This is of critical importance when the service is running on a public server, which may have hundreds or thousands of requests per second. The service will provide a means for the user level code to receive the contents of matching packets.

The Kernel Service functions cannot be called directly by user level code; instead, they are called as a result of operations on the service and device objects opened by code contained in the DLL. When the service is registered, and when the virtual "devices" which correspond to the

adapter capture interfaces are created, these functions are mapped to specific kernel operations. For example, when the Windows API function `StartService` is called, our kernel function `DriverEntry` will be invoked by the kernel. `DeviceOpen` will be the result of a `CreateFile` call, etc.

4.4 JAVASOCK SECURITY

The JavaSock API has a strong emphasis on security. The existing JavaSock API's security architecture makes use of the fine-grained security architecture available in Windows. The tightest level of security will involve the kernel level service, which intercepts incoming packets. During installation, which will require Administrator privileges on the machine, the service will be configured such that it can only be started by a user with Administrator privileges. This is performed using the Windows API function *SetServiceObjectSecurity*, or indirectly with an installation INF script. Similar security features are available on the Linux and Mac OS platforms, and will be used if the JavaSock API is ported to those platforms.

Subsequent connections to the service must originate from a user with sufficient access privileges. This helps guarantee that an orphaned service is not used by a third party for malicious purposes. Since Administrator privileges were required to start the service, the same process should still have them while connecting. A process may make several connections to the service to open different adapters for configuration and capture.

Once data capture is complete, the user level process should stop the service. This happens automatically if the shared library is unloaded (i.e., the calling process terminates). If this is not performed, the service will disable itself, preventing further accesses, after an appropriate interval of inactivity.

4.5 EXAMPLE APPLICATION

The following source code illustrates the use of the JavaSock API in a simple console based packet logging tool:

```

PacketListener pl = new PacketListener() {
    public void packetReceived(
        PacketHandler handler,
        java.nio.ByteBuffer data,
        java.util.List chain,
        PacketStatistics stats)
    {
        System.out.println(handler.getClass().getName());
        printPacket(data);
    }
};

OSIDataLinkDevice[] devices =
    OSIDataLinkDevice.getDevices();
OSIDataLinkDevice osld = devices[0];
osld.addPacketListener(pl);

IPHandler iph = new IPHandler(osld);

```

```

iph.addFilter(new IPFromToFilter(
    InetAddress.getLocalHost(),
    InetAddress.getByName("host.domain.com")
));
iph.addPacketListener(pl);

osld.startCapture();
System.in.read();          // Wait for user to press enter
osld.stopCapture();

```

Sample output from this test program might resemble something like:

```

org.javasock.EthernetDevice
78 bytes received:
00 0d 56 70 28 20 00 08 a1 22 35 f0 08 00 45 00
00 40 f4 4b 00 00 80 11 08 d7 0a 0b 22 5d 0a 0b
07 18 0a 25 00 35 00 2c ec 7d 74 d8 01 00 00 01
00 00 00 00 00 00 07 64 69 6c 62 65 72 74 06 63
72 65 61 72 65 03 63 6f 6d 00 00 01 00 01

```

```

org.javasock.IPHandler
64 bytes received:
45 00 00 40 f4 4b 00 00 80 11 08 d7 0a 0b 22 5d
0a 0b 07 18 0a 25 00 35 00 2c ec 7d 74 d8 01 00
00 01 00 00 00 00 00 00 07 64 69 6c 62 65 72 74
06 63 72 65 61 72 65 03 63 6f 6d 00 00 01 00 01

```

```

org.javasock.EthernetDevice
62 bytes received:
00 0d 56 70 27 10 00 08 a1 22 35 f0 08 00 45 00
00 30 f4 4d 40 00 80 06 c9 06 0a 0b 22 5d 0a 0b
07 01 0a 26 00 17 a5 17 08 c0 00 00 00 00 70 02
ff ff 8d 97 00 00 02 04 05 b4 01 01 04 02

```

```

org.javasock.IPHandler
48 bytes received:
45 00 00 30 f4 4d 40 00 80 06 c9 06 0a 0b 22 5d
0a 0b 07 01 0a 26 00 17 a5 17 08 c0 00 00 00 00
70 02 ff ff 8d 97 00 00 02 04 05 b4 01 01 04 02

```

These
represent
IP packets
from localhost.

5 FUTURE PLANS

The central objective of the proposed Phase II effort is to develop and enhance the JavaSock API, adding support to the API for writeable (outgoing) network connections, multiplatform support, and support for additional network devices and protocols.

To accomplish these objectives within the scope of the Phase II timeline and resources, we will focus on developing a fully functional API and demonstration applications on the Windows XP platform, while developing the multiplatform and additional network support

sufficiently to demonstrate JavaSock's full potential. In support of this goal, we have identified the following objectives:

1. Define the complete JavaSock API for read/write low-level network support. The JavaSock API developed in Phase I fully supports low-level network access in a read-only manner. During the Phase I effort, a draft extension to the API for including writeable packet support was written, but still remains to be implemented. Full implementation of writeable packets in the API is a cornerstone of the Phase II development, greatly increasing the usability and applicability of the JavaSock API.
2. Develop the JavaSock API on Windows. Once the API for writeable packet support is developed, the resulting API needs to be implemented. Since the Phase I JavaSock API was developed under the Windows operating system, the initial version of the Phase II JavaSock API will be developed on Windows as well.
3. Extend the JavaSock API to support additional platforms, network devices and network protocols. A key feature of the JavaSock API is that the majority of the library is implemented at the Java level, with only a minimal portion of the code implemented as native code (accessed through JNI). This approach maximizes security (the native code is limited to a small, easily reviewed library, while the majority of the code runs under Java, retaining Java's strong security model), while also allowing extendibility and modular design. The JavaSock library can be ported to other Java-supported platforms by rewriting the small native kernel portion (the Java portion remains platform independent). Similarly, the JavaSock API can be extended to support additional network devices (802.11(b/g) networks, DSL, Bluetooth, etc.) through simple modifications to the kernel service, and additional network protocols (ARP, ICMP, JREAP,¹ SCPS,² etc.) by simply extending the existing JavaSock network classes and methods.
4. Demonstrate the JavaSock API and demonstration applications. Once the API is designed and implemented, demonstration applications can be written to test the API implementation, demonstrate the capabilities of the JavaSock API, and serve as example source code for other development efforts.

¹ Interoperability Standard for the Joint Range Extension Application Protocol (JREAP), Mil-STD-3011, Sept. 30, 2002.

² SCPS—Space Communication Protocol Standard, <http://www.scps.org/>.